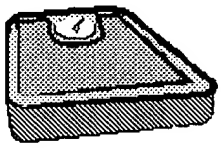
Mixing heavy and light components

By Amy Fowler

Although the Swing component set provides alternatives to using pre-Swing AWT components (such as Button, List, and the like), one of the primary design goals for Swing was that it be based on AWT architecture.



A significant benefit of such layering is that it greatly facilitates porting AWT-component programs to Swing. In fact, since both the AWT and Swing component sets use the same AWT infrastructure, it's possible to mix both kinds of components in the same program -- a technique that allows for phased migration of applications.

This article will lay out the rules for mixing components, and will provide some information and guidelines that we hope will make the mixing of AWT and Swing components as painless as possible.

Summary

NOTE: If you aren't familiar with AWT components or their architecture, see the "Introducing Swing" article in the current issue.

Heavy and light components

Most of the issues related to mixing AWT and Swing components are related to the mixing of so-called *heavyweight* and *lightweight* components. A *heavyweight* component is one that is associated with its own native screen resource (commonly known as a *peer*). A *lightweight* component is one that "borrows" the screen resource of an ancestor (which means it has no native resource of its own -- so it's "lighter").

(Lightweight component support was introduced in JDK1.1, and you can read more about it in the JDK1.1 Lightweight UI Framework design document.)

We generally don't recommend mixing Swing and AWT components because there are significant benefits in sticking with programs that are written entirely in Swing (and thus use only lightweight components).

Some of the benefits of using Swing components are:

- *More efficient use of resources*: Lightweight components are really "lighter" than heavyweight components.
- *More consistency across platforms* because Swing is written entirely in Java.

- *Cleaner look-and-feel integration*: You can give a set of components a matching look-and-feel by implementing them using Swing.

Despite the benefits of using Swing components exclusively, a developer may sometimes have to mix AWT components and Swing components in the same program (even when migration is not to blame). For example, such mixing may be required when a Swing version of a particular AWT component is not yet available.

Because there's sometimes no alternative to mixing heavyweight and lightweight components, we have provided a few options in Swing to make a certain level of component-mixing possible. However, as anyone who has tried this approach knows, there are some practical limitations to this approach.

Heavy vs. light: the differences

There are some significant differences between lightweight and heavyweight components. And, since all AWT components are heavyweight and all Swing components are lightweight (except for the top-level ones: JWindow, JFrame, JDialog, and JApplet), these differences become painfully apparent when you start mixing Swing components with AWT components.

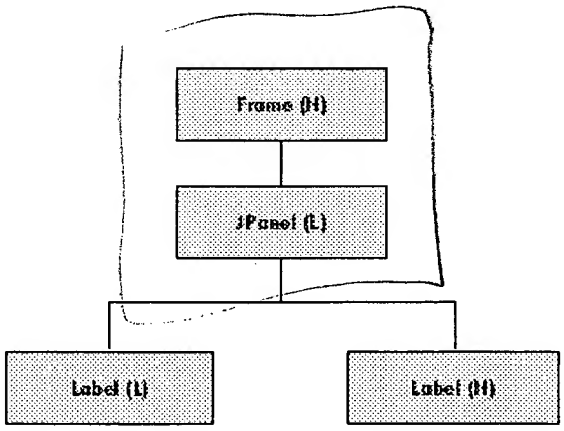
The differences boil down to the following:

- A lightweight component can have transparent pixels; a heavyweight is always opaque.

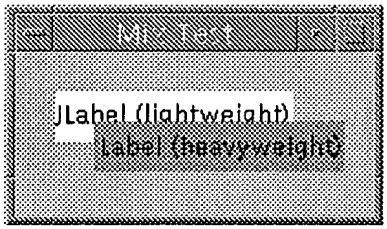
- A lightweight component can appear to be non-rectangular because of its ability to set transparent areas; a heavyweight can only be rectangular.
- Mouse events on a lightweight component fall through to its parent; mouse events on a heavyweight component do not fall through to its parent.
- When a lightweight component overlaps a heavyweight component, the heavyweight component is always on top, regardless of the relative z-order of the two components.

Z-order limitations

It is Issue No. 4 -- the one that applies to overlapping components -- that causes the most grief for developers who are mixing AWT and Swing components. This limitation exists because of the fact that a lightweight component re-uses the screen real estate of its nearest heavyweight ancestor and is therefore restricted to the z-order position of that ancestor. To illustrate, say your application had the following containment hierarchy (where "H" means heavyweight and "L" means lightweight):



For the sake of the example, let's assume that the two labels overlap, and that the z-order value of the Swing label places it "on top" of the AWT label. But it still looks like this:



The Swing label will never appear to be on top of the AWT label because the Swing label is rendering its contents in the native window of the frame (its first heavyweight ancestor), while the AWT label is rendering its contents in its own native window, which is actually a child of the frame's native window.

If the Swing and AWT components do not overlap, then in general there should be no problems mixing them within the same container.

Guideline No. 1

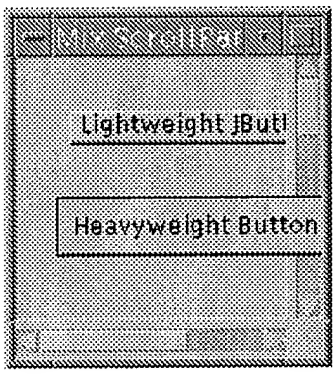
Do not mix lightweight (Swing) and heavyweight (AWT) components within a container where the lightweight component is expected to overlap the heavyweight one.

Swing-specific Z-order issues

The lightweight/heavyweight z-order limitation causes some problems in specific areas of Swing. When developers run into these problems, they often don't realize the cause, and immediately conclude that there's a bug in Swing. The three most common z-order-related problems have to do with Swing's JScrollPane and JInternalFrame classes, and its popup components.

Swing's ScrollPane

Both the JScrollPane class and its viewport -- which clips its scrollable child -- are lightweight components. Consequently, heavyweight components cannot be handled properly when they are placed inside a JScrollPane's viewport. This is a window created by a little program that places both a Swing JButton and an AWT Button within a JScrollPane:



As you can see, the lightweight viewport cannot -- and does not -- clip the heavyweight AWT button properly.

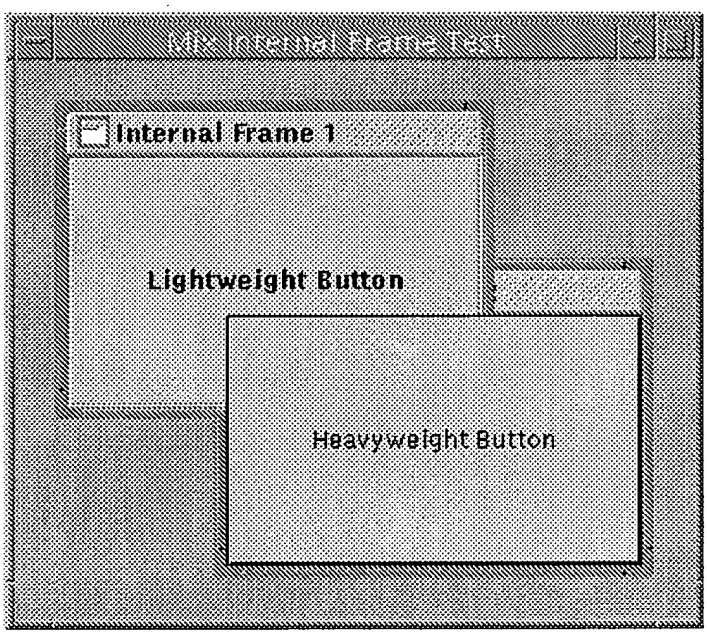
Guideline No. 2

Do not place heavyweight (AWT) components inside a JScrollPane. If you need to scroll areas containing heavyweight

components, use AWT's ScrollPane instead.

Swing's JInternalFrame component

Swing's JInternalFrame is a lightweight component. Typically, InternalFrame components are placed inside a JDesktopPane component, where the user can move the InternalFrames around in such a way that they can potentially overlap. Therefore, if heavyweight components are placed inside an InternalFrame, and that InternalFrame is overlapped by another InternalFrame, the first frame's heavyweight children will always be on top, potentially obscuring the other InternalFrame. This is a screen shot that illustrates this problem:



The Swing team is considering adding the option of creating InternalFrames as heavyweight containers in a future Swing release to alleviate this problem. However, until then, adhere to the following guideline:

Guideline No. 3

Do not place heavyweight (AWT) components inside a JInternalFrame.

Swing popup components

Swing provides a number of components that create *popup windows* (windows that dynamically appear when the user clicks on the component). These popup components are JPopupMenu, JComboBox, and JMenuBar.

Swing attempts to maximize popup performance by dynamically choosing the most efficient container to display a popup just prior to showing the popup. The types of popups created are:

- *LightWeight Popup*: a lightweight container.
- *MediumWeight Popup*: an AWT heavyweight Panel.
- *HeavyWeight Popup*: a JWindow (heavyweight top-level window).

The rules that Swing uses to determine which popup to use are:

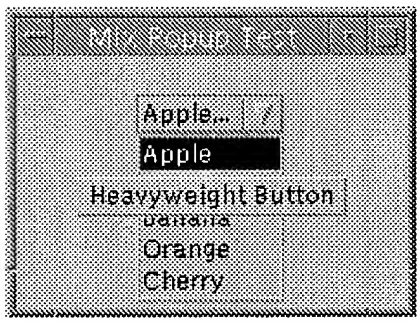
- If a popup does not fit within the bounds of the top-level window that contains the popup component, then a *HeavyWeight popup* is always used to ensure the entire popup will be visible.
- If a popup will fit within the bounds of the top-level window which contains the popup component, and a popup's

`lightWeightPopupEnabled` property is `true`, then a *LightWeight popup* is used for maximum efficiency.

- If a popup will fit within the bounds of the top-level window that contains the popup component, and a popup's `lightWeightPopupEnabled` property is `false`, then a *MediumWeight popup* is used.

Another benefit of displaying lightweight and medium-weight popups is that within a browser environment, this strategy avoids the display of the "Warning Applet" banner in the menus.

By default, the `lightWeightPopupEnabled` property is set to `true` for all popup components. Therefore, if you place a Swing popup component in a container that also contains an AWT heavyweight control, then it's possible that the popup will be obscured by the heavyweight control (if conditions are such that a "lightweight popup" is used). For example, if you place a JComboBox and an AWT Button in a panel, you might see the following result when you activate the combo box:



Notice that the combo box's popup window is painted *behind* the AWT button!

Fortunately, programs can easily disable the LightWeight Popup feature by setting the

`lightWeightPopupEnabled` property to `false`. There are two ways to perform this operation. First, you can change the default value for this property globally for a program using the following static method on JPopupMenu:

```
public static void
   setDefaultLightWeightPopupEnabled(boolean)
```

Once this method is invoked with a value of `false`, all popup components subsequently instantiated will initialize their lightWeightPopupEnabled property to false.

Second, a program can control this for individual popup instances by using instance methods provided by JPopupMenu and JComboBox:

```
public void setLightWeightPopupEnabled(boolean
```

Setting this property on individual popup instances will override the default value controlled by the static JPopupMenu method.

Example: Mixing components

The following sample program mixes a Swing MenuBar (a lightweight component) with an AWT Button (A heavyweight component). The "Lite" menu-bar menu defaults to enable lightweight popups while the "Heavy" menu uses the `setDefaultLightWeightPopupEnabled()` method to ensure the menu is never lightweight.

```
import java.awt.*;
```

```
import com.sun.java.swing.*;

public class MixPopupTest extends JFrame {

    public MixPopupTest() {
       super("Mix Popup Test");

       JMenuBar menubar = new JMenuBar();
       setJMenuBar(menubar);

       // Create lightweight-enabled menu
       JMenu menu = new JMenu("Lite Menu");
       menu.add("Salad");
       menu.add("Fruit Plate");
       menu.add("Water");
       menubar.add(menu);

       // Create lightweight-disabled menu
       JPopupMenu.
          setDefaultLightWeightPopupEnabled
          (false);
       menu = new JMenu("Heavy Menu");
       menu.add("Filet Mignon");
       menu.add("Gravy");
       menu.add("Banana Split");
       menubar.add(menu);

       // Create Heavyweight AWT Button
       Button heavy = new Button
           ("  Heavyweight Button  ");

       // Add heavy button to box
       Box box = Box.createVerticalBox();
       box.add(Box.createVerticalStrut(20));
       box.add(heavy);
       box.add(Box.createVerticalStrut(20));
       getContentPane().add("Center", box);

       pack();
    }
    public static void main(String[] args) {
        MixPopupTest t = new MixPopupTest();
```

```
            t.show();
        }
    }
```

The Solaris version of AWT currently has a bug that can cause MediumWeight popups to not be displayed properly. See the Solaris MediumWeight popup bug overview for a more complete description of the problem and the workaround you can use to fix it.

Guideline No. 4

If you place Swing popup components in a window containing heavyweight components and it's possible that the popup windows will intersect a heavyweight, then invoke

```
JPopupMenu.setDefaultLightWeightPopupEnabl
    (false)
```

before the popup components are instantiated.

Summary

If you understand the limitations of mixing Swing and AWT components, it is generally possible to build Java programs which use both in harmony. However, because there are significant advantages to using lightweight components, we recommend using lightweights (and Swing!) wherever possible and only using heavyweight components where a lightweight version either isn't yet available or doesn't meet your

program's needs.

Download Swing ▫ Swing API Documentation ▫ JDK Documentation
JDK Download ▫ JFC Web Site ▫ README

The Swing Connection -- Volume 3, No.4
Swing Version 1.0. Last modified 2/98.